**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 |  | 实验时间 | 2023年 12月 8日 |
| 诚信声明 | 手写，签名 | | |
| 实验题目 | 简单排序方法的实现与比较 | | |
| 实验过程中遇到的主要问题 | * 在读取文件并将整数读入数组时，需要处理可能出现的读取错误。 * 在计算排序过程中的比较次数、交换次数以及所消耗的时间时，为了精确地测量这些参数，要使用C++中的时间函数。 | | |
| 实验小结 | 本次实验我实现了冒泡排序、简单选择排序和直接插入排序三种简单排序方法，并对它们的性能进行了比较。我发现，尽管在理论上这三种排序方法的时间复杂度都是O(n^2)，但在实际应用中，它们的性能差距还是很大的。  具体来说，冒泡排序的性能最差，因为其需要进行大量的数据交换和比较；而简单选择排序和直接插入排序的性能要好一些，简单选择排序的交换次数较少，而直接插入排序的比较次数较少。此外，我还发现，直接插入排序在数据近乎有序的情况下，性能可以接近O(n)，这使得它在某些特定情况下比其他两种排序方法更有优势。 | | |
| 数据结构  （自定义数据类型） | #define MAXN 50000  // 使用数组存储顺序表  int \*a = new int[MAXN];  // 存储排序进行的运算次数  struct SortResult {  int comparisons;  int swaps;  }; | | |
| 主要算法  （或算法说明） | 冒泡排序：从数组的开始，比较每一对相邻元素，如果它们的顺序错误就把它们交换过来。这样，每一轮过后，最大的未被排序的元素就会被交换到数组的最后。记录最后一次交换的位置作为下一次的终点，重复这个过程，直到整个数组都排序好（上一次排序未交换）。  简单选择排序：在未排序的数组中找到最小的元素，将其存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。  直接插入排序：从数组的第二个元素开始，将其与前面已排序的数组依次比较，来插入到前面已排序的数组中正确的位置。重复这个过程，直到整个数组都排序好。 | | |

**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 |  | 实验时间 | 2023年 12月 8日 |
| 诚信声明 | 手写，签名 | | |
| 实验题目 | 堆排序的实现 | | |
| 实验过程中遇到的主要问题 | * 在生成随机数的过程中要保证生成的随机数具有足够的随机性，以便于正确测试堆排序算法的平均情况的效率。 * 在测量排序过程所消耗的时间时，需要保证测量的准确性。因为堆排序的时间复杂度是O(n\*log(n))，所以当处理的数据量不够大时，时间测量的误差可能会影响到实验结果的准确性。 | | |
| 实验小结 | 本次实验我成功地使用堆排序算法对我随机生成的20000个整数进行了排序，同时也测量了其平均性能。在排序过程中，我记录了元素的比较次数、交换次数以及排序过程所消耗的时间。我验证了，堆排序的效率确实满足其时间复杂度O(n\*log(n))的理论预期，证明了堆排序在处理大量数据时的优越性。这次实验使我们更深入地理解了堆排序的原理和性能。 | | |
| 数据结构  （自定义数据类型） | #define MAXN 50000  // 使用数组存储堆  int \*a = new int[MAXN];  // 存储排序进行的运算次数  struct SortResult {  int comparisons;  int swaps;  }; | | |
| 主要算法  （或算法说明） | 堆排序：  堆排序是一种树形选择排序，是对直接选择排序的有效改进，步骤如下。   1. 将待排序的序列构造成一个大顶堆（正序从小到大排列）。此时，整个序列的最大值就是堆顶的根节点。 2. 将根节点与最后一个元素交换位置，然后从堆中删除最后一个元素。此时，得到的新的根节点可能违反堆的性质，因此需要对当前的根节点使用筛选算法，使之成为新的大顶堆。 3. 重复步骤2，直到堆中只剩下一个元素。 | | |